Parametric Polymorphism Optimization for Deeply Nested Types in Computer Algebra

Laurentiu Dragan Stephen M. Watt

Ontario Research Center Computer Algebra
University of Western Ontario
Department of Computer Science
London, Ontario, Canada N6A 5B7

{ldragan,watt}@scl.csd.uwo.ca

Abstract

Computer algebra systems, such as Axiom, and programming languages designed for computer algebra, such as Aldor, have very flexible mechanisms for generic code, with type parameterization. Modern versions of Maple can support this style of programming through the use of Maple's module system, and by using module-producing functions to give parametric type constructors. From the software design point of view, generic programming allows better code re-usability so main-stream programming languages, such as C++ and Java, have evolved to support it.

Computer algebra programs that use parametric polymorphism tend to do so very heavily, leading to deeply nested type constructions. Optimization of deeply nested type constructions thus becomes an important problem for system efficiency. The goal of the current work is to present optimizations for this situation. Our approach is to specialize generic types at compile-time, based on global program analysis. We anticipate that the problem of optimizing deeply nested type construction will also find application outside computer algebra.

Keywords: Parametric polymorphism, generics, code optimization, specialization, Maple, Aldor.

1. Introduction

As computer algebra systems become more complex, it becomes increasingly important to adopt design strategies that help make development as manageable as possible. The complex interactions among mathematical library components can lead to problems with correctness and efficiency. One strategy that has been adopted to improve the scope of code re-use is the notion of generic libraries that may be instantiated for specific types, providing what is known as parametric polymorphism. This is a particularly powerful notion in the mathematical context, where rich relations exist among algebraic types, and where strong guarantees can be made about the relationship among parameters.

Many general programming languages, such as Aldor, Modula-3 and C++, allow the programmer to use parametric polymorphism. Newer languages, like Java and C#, recognize the value of parametric types and have provided retro-fitted implementations. Specialized programming languages such as Maple can also make use of generic

modules/types to develop generic algorithms [1]. This style of Maple programming can be quite flexible, as is the one we see in Aldor programming language.

In Aldor, types are represented by abstract data types known as *domains*. These are run-time values, belonging to *type categories* that specify mathematical properties and allow compile time optimization. Maple has simulated this behaviour with `Domains` package, which allows users to program in the style of Axiom/Aldor. Modern versions of Maple provide modules, which present an improved implementation for domains in Maple. Parametric modules can be produced by module-producing procedures, effectively creating parametric types in Maple. Additionally, it is possible to use the Alma [2] system to connect these two approaches. Alma allows Aldor's efficiently compiled domain constructors to appear in Maple as module-producing functions. We thus see that efficiency improvements in parametric polymorphism can benefit Aldor, Maple and their combination.

The main goal of this work is to see how specialization can improve the performance of deeply nested generics for computer algebra. We first examine this in the context of Aldor's performance. This improvement can also be used to generate more efficient libraries for Maple when using Aldor through the Alma system. We also examine the potential for improvement in specializing Maple's generics, by hand-specializing representative code. It would be useful to write a partial evaluator for Maple programs and the be able to optimize them in a similar fashion to what we propose here for Aldor, but this require more work in Maple to implement the optimization tools that are already present in the Aldor compiler.

In computer algebra systems, scientific libraries and other systems that have rich mathematical models, it is possible to write very general algorithms. When generics are heavily used, deeply nested types can be formed. An example of such a nested type is:

```
List(Matrix(Poly(Complex(DoubleFloat))))
```

Because the Aldor programming language supports generic types in a natural way, and because the type system of Aldor allows easy manipulation of types, we decided to conduct our experiments using the framework provided by Aldor. So far, we have implemented such a specializer for the Aldor compiler at the intermediate language level. The results obtained are encouraging.

2. Parametric Polymorphism in Different Programming Languages

2.1. Heterogeneous and Homogeneous Implementation

In the recent years, we have seen an increase in widespread acceptance of parametric polymorphism. This feature has already been present in languages like Modula-3, ADA and C++ for some time. Java has retro-fitted this feature in its most recent version, and the next release of the C#/.NET platform will also support generic programming.

Using parametric polymorphism without performance penalty (or with a small overhead) is very important. With the introduction of generics in the main programming languages we expect to see more use of generic style programming. We therefore expect our work to have increasing impact as parametric polymorphism becomes more widely adopted..

Currently, there are two approaches to implement the parametric polymorphism paradigm: *homogeneous* and *heterogeneous*.

The *heterogeneous* approach constructs a special class for each of the type parameters. For example, in case of the `Vector` from C++ STL, one can construct `Vector<int>` or `Vector<double>`. Because C++ uses heterogeneous approach, two distinct classes are generated for the above cases: `VectorInt` and `VectorDouble`. They duplicate the source code of the `Vector` generic class and produce, different *specialized* compiled forms of the vector class.

This is a good model from the time efficiency point of view, because the two classes are working on their own form of the code, which can be optimized since it is not parameterized any more. The drawback of this method is the size of the code. For each instance there is a copy of the code.

The *homogeneous* approach uses the same generic class at runtime for every instance of the type parameters. This method is used in Java by erasing the type information and using the Object class, instead of the specialized form, and by casting back to the target class whenever necessary. This method has small run-time overhead and the code size is not increased from the original one. For example, `Vector<Integer>` will be transformed to a `Vector` that contains `Object` class objects, and the compiler will check if an `Integer` class object is used when referring to elements of the vector. This assures that same code is used for `Vector<Pineapple>`.

Aldor is uses the homogeneous approach, but the Aldor implementation is in many ways superior to the one we see in Java. In Aldor the types are not erased, and this provides very good flexibility and the ability to implement domains in a lazy manner. It also allows for generics to be instantiated differently according to run-time decisions. The downside of this implementation is that it does not achieve the same execution speed as the heterogeneous approach, unless code optimizations are performed. This is even more visible when the parameters of the domains are themselves parametric domains, leading to *deeply nested* generics.

The .NET platform implements generics in a homogeneous manner at the intermediate language (CLR) level. In .NET, the type information is preserved by the compiler and is used at run-time by the just-in-time compiler to optimize the code.

2.2. Parametric Polymorphism in Maple

It is possible to achieve parametric polymorphism in Maple using modules as types and module-producing functions as type constructors. One simple example can be seen in the following:

```
# Integers mod p.
IMod := proc (p)
     module()
         export zero, `+`, `*`, `=`, convertIn, convertOut;
         zero:= 0;
         `+` := (a,b) -> a+b mod p;
         `*` := (a,b) -> a*b mod p;
         `=` := (a,b) -> evalb(a = b);
         convertIn := proc(n)
             if not type(n, 'integer') then error("Bad number") end if;
             n mod p
         end;
         convertOut := v -> v;
     end module
end proc:
```

```
# Poly(x)(R) gives polynomials in the variable x with coefficient ring R.
Poly := proc (x) proc(R)
     module()
         export zero, `+`, `*`, `=`, convertIn, convertOut, pcoef, pdegree;
         local fixDegree;
         zero := [];
         # Drop leading coefficients equal to zero.
         dropZeros := proc(p)
              local i, dtrue, d;
              d := pdegree(p);
              dtrue := -1;
              for i from d to 0 by -1 do
                  if not R:-`=`(R:-zero, pcoef(p,i)) then
                      dtrue := i;
                      break
                  end if
              end do;
              if d = dtrue then p else [op(0..dtrue+1, p)] end if
         end proc:
         `+` := proc(p,q)
              local i, d;
              d := max(pdegree(p), pdegree(q));
              dropZeros([seq(R:-`+`(pcoef(p,i), pcoef(q,i)), i=0..d)]);
         end proc:
         `*` := proc(p, q)
             local a, i, j, pd, qd;

             if p = zero or q = zero then return zero end if;

             pd := pdegree(p); qd := pdegree(q);
             a := array(0..pd+qd);
             for i from 0 to pd + qd do a[i] := R:-zero(); end do;
             for i from 0 to pd do
                 for j from 0 to qd do
                     a[i+j] := R:-`+`(a[i+j], R:-`*`(pcoef(p,i), pcoef(q,j)));
                 end do
             end do;
             [seq(a[i], i = 0..pd+qd)]
         end proc:
         `=` := proc(p,q)
             local i;
             if pdegree(p) <> pdegree(q) then return false end if;
             for i from 0 to degree(p) do
                 if not R:-`=`(pcoef(p,i), pcoef(q,i)) then return false end if
             end do;
             true
         end proc:
         convertIn := proc(w0)
             local i, w;
             w := collect(w0,x);
             dropZeros([seq(R:-convertIn(coeff(w,x,i)), i=0..degree(w,x))])
         end proc;
         convertOut := proc(p)
             local i, dp;
             dp := pdegree(p);
             add(R:-convertOut(pcoef(p,dp-i)) * x^(dp-i), i = 0..dp)
         end proc;
         pdegree := p -> nops(p) - 1;
         pcoef := proc(p,i)
             if i+1<=nops(p) then p[i+1] else R:-zero end if
         end proc:
     end module
end proc end proc:
```

```
# Curry the constructors to make Ring to Ring functors.
Px := Poly(x);
Py := Poly(y);
Pz := Poly(z);

# Compose a type tower of 4 nested constructors.
pm := Px(Py(Pz(IMod(17))));

# Perform some polynomial arithmetic.
a0  := 2*x^2+4*y^2+3*z^2+(2*x*y*z)+y*(4*x+5*z);
a   := pm:-convertIn(a0);
oa  := pm:-convertOut(a);
o2a := pm:-convertOut(pm:-`+`(a,a));
oaa := pm:-convertOut(pm:-`*`(a,a));
```

In this example we create a parametric type constructor, `IMod`, that produces a ring of integers modulo n. The parameterized constructor `Poly` produces a polynomial ring using a dense representation. `Poly` takes a symbol and returns a constructor requiring a coefficient ring. `IMod(n)` and `Poly(x,R)` both yield such objects. Unfortunately, in Maple is not possible to easily specify type of this kind of type parameter. Both of our constructors produce types that export the functions `=,+,*,convertIn, convertOut` and the `Poly` constructor relies on these being provided by the parameter giving the coefficient ring. This example shows one way to write generic algorithms in Maple.

2.3. Parametric Polymorphism in Aldor

Aldor was initially designed as an extension programming language for Axiom computer algebra system. At present, it can be used as a general-purpose programming language, but its main application remains computer algebra, as can be seen from its extensive algebra libraries. In Aldor, functions and types are first-class values, allowing great flexibility for types. This is allows elegant support the complex relationships between mathematical structures.

The type system in Aldor is organized on two levels: domains and categories. Categories are used to represent the type of domains. More details about Aldor programming language can be found in Aldor User Guide [3].

The concept of domains and categories is similar to ideas in object-oriented systems, where domains are the classes and categories are the interfaces. However, they are not exactly the same since domains do not allow inheritance. The main advantage of Aldor's types system is that is allows the type checking to verify whether an object belongs to a type rather than belonging to some unknown sub-type of that type.

In Aldor, dependent types are used to place restrictions on parametric polymorphism, similar to the bounded polymorphism that is present in Java and C#. An example of a dependent type is `(n:Integer, m:integer)->IMod(m),` the type of functions taking two integers as arguments and returning a result in the type of integers modulo the second argument. This shows how dependent types can be used in programs where the type of a result depends on the value of an argument. Dependent types are particularly useful in type producing functions, e.g. `(R: Ring) -> Module(R).`

In order to see how to use Aldor domains, consider the following small example of a parametric type declaration:

```
 1. import from SingleInteger;
 2. define Dom1Cat: Category == with{m: SingleInteger->SingleInteger;}
 3.
 4. Dom1: Dom1Cat == add {m(g: SingleInteger):SingleInteger==g := g+1;}
 5.
 6. Dom2(p: Dom1Cat): Dom1Cat == add {
 7.    m(x: SingleInteger) : SingleInteger == {
 8.         for i in 1..1000 repeat x := m(x)$p + 1;}
 9.         x;
10.     }
11. }
12.
13. import from Dom2 Dom1;
14.
15. print << m(0) << newline;
```

In this example, `Dom1Cat` is a category. This category defines a type and any domain that has this type should implement the function `m`. This is very similar to interfaces from object-oriented languages such as Java, where interfaces are used to specify the functions that should be implemented by the `implementing' class.

Later in the example, a domain `Dom1` is defined. `Dom1` is declared to be of type `Dom1Cat`. This means that `Dom1` *implements* `Dom1Cat`, and as a consequence, `Dom1` *must* provide the code for program `m`.

Next, a new domain is defined `Dom2`. It can be seen that `Dom2` is parameterized, and the type of the parameter is `Dom1Cat`. `Dom2` has the type `Dom1Cat`, which means it should implement `m`. As can be seen from the above declaration, `Dom2` is a function that takes a domain as a parameter, and returns a domain as its result. The code for the function is a domain generating expression that uses the parameter `p`. It can also be seen that the parameter is a domain of type `Dom1Cat`. This puts a restriction on the possible parameters, so this parametric polymorphism is *bounded*. In C++ it is not possible to specify the type of the parameter, and therefore type checking is done partially at link-time for C++.

Lines 13, 15 show how `Dom2` can be used. In this example, the function `m` from domain `Dom2` is called.

One of the strengths of the Aldor programming language is that functions and types are first-class values. This way, functions and types can be manipulated in the same way as any other values; they can be assigned to variables or returned from functions. This is why, in the above example, a generic domain can be given by a simple function that constructs a new domain based on its parameter.

3. Deeply Nested Types

Using a generic library for computer algebra, such as the one provided by Aldor, one can easily find one is working with objects of a complicated type, such as

```
List(Matrix(Poly(Complex(DoubleFloat))))
```

This construction tells us that we have a list whose elements are matrices; the elements of the matrix are polynomials with complex coefficients. Also, the real and imaginary parts of complex number are implemented using floating-point numbers. This is an example of a *deeply nested type*.

Because `DoubleFloat` is not a parametric domain, the type `Complex(DoubleFloat)` is not parametric. Using same reasoning `List Matrix Poly Complex DoubleFloat` is not a parametric type.

Now, let us assume that one would call an operation from the `List` domainr, for example, to map multiplication by a constant onto each element. This means each element of the list should be multiplied with the constant. But the elements are matrices, which means call a function (to multiply a matrix by a constant) from the `Matrix` domain. Each element from the matrix is a polynomial, which requires invoking the constant multiplication operation from the polynomial. This then invokes polynomial coefficient operations, etc.

This kind of operation requires many function calls whenever a function from a domain is called. This introduces an overhead that can be avoided by *specializing* the domain. By specializing the operations of the domain, it might be possible further optimize the resulting operation by constant propagation and dead-variable elimination optimizations.

After constructing the domain presented in the above example, only operations from `List` are directly invoked in the program. This gives us the idea of constructing a *specialized* domain:

```
List o Matrix o Poly o Complex o DoubleFloat
```

The functions of this domain would be specialized forms of the functions from all of the domains. We do not need partial in-lining since the calls will invoke only operations from `List`. Creating a specialized function with code from in-lined functions can benefit from other optimization techniques that are not inter-procedural.

Since the Aldor language allows dynamic types, we may not know a component of the deeply nested domain at compile-time. We may have situations such as

```
List Matrix Poly Complex           X
X    Matrix Poly Complex DoubleFloat
List Matrix X    Complex DoubleFloat
```

where `X` is unknown at compile-time. These cases require special care because we cannot in-line code from X.

As stated earlier, the Aldor programming language uses at the moment a homogeneous approach for parameterized domains. All operations on objects of the parameter type are performed via function calls. This makes the resulting code flexible, but slower. Our experiment implements the heterogeneous approach in cases of deeply nested domains. Using the Aldor compiler, we try to specialize the parametric type at compile-time, when possible. This is similar to the heterogeneous case seen in C++. The difference from C++ is that, in Aldor, the code specialization is not done by a simple macro substitution. Instead it is done by specializing the domains after the type checking system has verified that the program is type safe. In some cases, the code size increase is more important than the execution speed. In these cases, the homogeneous approach may be desired. In the future, it would be interesting to investigate the use of heuristics, to decide when to specialize.

4. Code Specialization

The main idea for this optimization is to expand in-line the code of the called function from the parametric domain in the caller function. Using this optimization we benefit from eliminating the dispatch overhead, and also from in-lining the code of the called function. This inlining allows further optimization, by making the inlined function's code visible to the local optimizations of the caller function. For example this often allows additional copy propagation, common sub-expression elimination and dead code elimination.

Inlining in Aldor must take care of the usual cases of creating temporary variables when arguments are complex, potentially side-effecting expressions, and when the inlined function appears in the middle of an expression. Additionally, because Aldor is a functional language supporting closures, inlining must manipulate closure objects so that the inlined code accesses its lexical variables from the correct environments.

This idea of specialization is related to the partial evaluation technique were the code is specialized for a particular input type [4]. Partial evaluation can be used to specialize generic algorithms. Generic programming has the advantage of being easy to maintain, but does comes at the cost of efficiency. One solution would be to write specialized code by hand, but this hard to do in practice and error prone. This is why partial evaluators were written. A partial evaluator is a tool that transforms the generic code into a specialized code based on the constant inputs used by the generic code. The constant values are usually application dependent and are provided by the user. This tool is used for scientific computing, computer graphics, and it can yield good improvements over the generic code. A partial evaluator only takes the constant inputs and propagates them through the program. The difference in our case is that we analyze the whole program first and find the particular types that were created by the programmer and perform an *automatic* specialization based on that type.

We intend to experiment with partial specializations based on program particularities. Dean, Chambers and Grove did similar work in their selective specialization [5]. For object-oriented languages a problem arises when the call is virtual and a dynamic dispatch is required. The specialization can be done on all possible types, but that would result in over-specialization. To avoid this, in [5], a profile based specialization is proposed, that would specialize only the calls that are most beneficial. Also, [5] specializes only the messages between objects, not the parametric types.

We have implemented a specializer for Aldor framework. In the current state it performs a full type specialization more like C++ templates.

5. Optimization for the Aldor Framework and Results

In this section, we shall present the optimization implemented in Aldor, and the results obtained so far.

The Aldor compiler supports many operating systems. In order to be able to achieve this, the compiler uses a portable intermediate language representation. The intermediate language is called FOAM and it stands for First Order Abstract Machine [6]. The goal of FOAM is to be platform independent, to have efficient mapping to C and Lisp and to allow easy manipulation. The quality of easy manipulation is useful for FOAM-to-FOAM optimization. Our code transformation is performed at the FOAM level.

5.1. Code Specialization for Aldor

In our case we use a partial evaluator that not only specializes the domain constructor, but also specializes all the exports of that domain. This creates operations of that specialized domain as monolithic operations that are more efficient.

Some complications arise in Aldor programming language, since the domains are dynamic, created at runtime. Any information related to domains is therefore known only at runtime.

Although some domains may be determined completely as run-time values, compiling whole program permits an overview of the whole code, providing visibility to those complete type forms that areknown at compile time.

This is not the case when libraries are generated, because the parametric domain may not be used (instantiated). A library would creates generic domains that will be used later by another program. But even in this case some domains may contain some partial instantiations of other parametric domains, and this should be optimized by our optimizing technique.

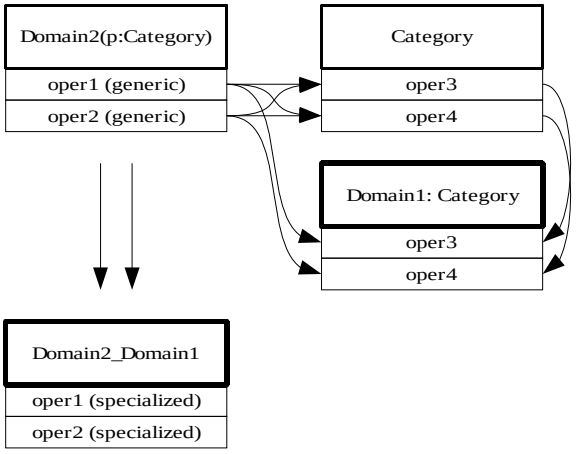

Figure 1. Domain specialization. Calls to generic domains are replaced by a specialized domain that does not require external calls.

To make the process more clear, let us turn to our example and see how it works. In our example, there are two domains: `Dom1` and `Dom2(.)`. They both belong to the same category. `Dom2` is a domain constructor, which accepts a parameter of type `Dom1Cat` and returns a domain of type `Dom1Cat`. In this case, a construction of type `Dom2(Dom1)` is possible. Our optimizer will detect the generic type construction and try to specialize it by replacing it with `Dom2_Dom1`, which will contain the specialized form of the operations (see Figure 1).

The proposed method will try to create specialized forms of the functions of instantiated domains. And use the specialized forms to construct the specialized run-time domains. The specialized functions should in-line code from domains that are type parameters. This way the resulting domains will have functions that do not require function call overhead when executing the code from a parameter domain.

Information about domains should always be available to executable programs because everything that is used must be fully declared, or inferred, at compile time. So, by analyzing the code, all syntactic instances can be discovered. These domain expressions may use some other domains or parameterized constructors that are not fully defined. In this case, a partial optimization can be applied by specializing only the part of the domain that is known at compile time.

In case of a library, or when Aldor's interpreted mode is used, some partial definitions may be encountered, as is the case for domains, but because the main function is not provided some of the declarations may be left in a parametric form. In this case, a partial optimization would be constructed based on the provided domains.

By partial optimization, we mean that if a declaration such as `Dom3(Dom2(Dom1))` is given, and only `Dom2` and `Dom1` are known or only `Dom3` and `Dom2` are known, we can specialize only `Dom2(Dom1)` or `Dom3 o Dom2` respectively.

The algorithm used to transform the FOAM code is:

1. initialize the data structures
2. identify the domain declarations
3. for each program
   3.1. if there is a domain constructor, generate a new specialized domain based on the domain constructor
   3.2. replace the call to the generic domain to a call to the specialized one
   3.3. in the specialized domain try to find the imports from other domains, and if found, modify the FOAM representation to help Aldor in-liner identify the called function.
4. construct the FOAM code back from the data structures used by the tower optimizer

To find new domain constructors, the program is scanned until a variable of type domain is defined, then the constructor is analyzed and the new domain is created.

To replace the call to the generic domain with a call to the specialized one, a new global variable is defined and initialized with the new domain and then the new value is assigned to the local variable that should be defined with the domain.

Finding the imported symbols from domains required functions to evaluate hash functions in FOAM code to simulate the run-time look up of domain exports.

By using this inter-procedural optimization, we could create specialized versions that contain constants in places of parameters. As such, we could leave to the other intra-procedural optimization techniques further code simplification. For example, by using constant propagation and dead variable elimination optimizations:

```
f(a:AldorInteger):(AldorInteger) == {
       b: AldorInteger;
       if (a > 10) then {  b := 0; for i in 1 .. a repeat b := b + 2 * i; }
       else b := 10;
       a + b;
}
```

If we call the function f(5) then we could optimize the function by propagating the constant:

```
f():(AldorInteger) == {
       b: AldorInteger;
       if (5 > 10) then {  b := 0; for i in 1 .. 5 repeat b := b + 2 * i; }
       else b := 10;
       5 + b;
}
```

By removing the if statement with the branch that is always taken, we get

```
f():(AldorInteger) == { b: AldorInteger; b := 10; 5 + b; }
```

and finally

```
f():(AldorInteger) == 15;
```

The compiler already performs some of these optimizations but they all work inside a function. To take advantage of them, we need to have a specialized domain. In some cases the compiler can detect the domain that is constructed, but in some others it cannot. So, for example if a function constructs two domains that are dependent of a variable parameter it is not possible for the compiler to specialize the function at compilation time, but it might be possible to generate the two possible *specialized* and *optimized* domains and use one of those depending on the input. Of course this would produce bigger code, but it will produce faster one at the same time.

6. Experiments

6.1. Aldor

To test type the potential for tower optimizations, we first created test files and specialized them manually at the Aldor language level.

The test file has the following structure:

```
#include "aldor"
#include "aldorio"
macro {LOOPS == 300; TESTS == 20;}
int ==> MachineInteger;
import from int;

define Dom1Cat: Category == with { m: int -> int; };
define Dom1: Dom1Cat == add { m(g: int) : int == g := g + 1; }

define Dom2(p: Dom1Cat): Dom1Cat == add
 { m(x: int): int == {for i in 1..LOOPS repeat {x := m(x)$p + 2;} x;} }

define Dom3(p: Dom1Cat): Dom1Cat == add
 { m(x: int): int == {for i in 1..LOOPS repeat {x := m(x)$p + 3;} x;} }

define Dom4(p: Dom1Cat): Dom1Cat == add
 { m(x: int): int == {for i in 1..LOOPS repeat {x := m(x)$p + 4;} x;} }

Dom1Cat: Category == with {m: (SingleInteger) -> SingleInteger;};

import from Dom4(Dom3(Dom2(Dom1)));
m(1);
```

We present the results in Table 1 for the current implementation of our optimizer. The results are encouraging. We note that these examples are simple and with very small functions. These examples did little more than calling the functions from inner domains, so that means that the program is spending most of its time in function call overhead, which makes this benchmark favourable.

Test	***Unoptimized Time(s)***	***Optimized Time(s)***	***Speedup(%)***
Test3	3.81	0.26	93.17
Test4	12.63	2.17	82.81
Test5	0.25	0.25	0
Test6	12.72	2.00	84.27
Test7	15.49	13.81	10.84
Test8	15.62	14.25	8.77

Table 1: Results for the automatically optimized tests

The results for partial specialization are given in the Table 2. We see here that the most important specialization is the innermost domain `Dom2_Dom1`. This is intuitive, since the most called function is the one in the inner-most domain.

All specializations that contain this specialization of the inner domain exhibit a similar performance increase, so the outer specializations have little effect in this case. We also note that specialization of domain constructor composition, e.g.

```
(Dom4 o Dom3 o Dom2)(X),
```

had no beneficial effect.

Specialization	***Speedup(%)***
Dom4_Dom3 Dom2 Dom1	-1.997
Dom4_Dom3_Dom2 Dom1	-7.132
Dom4_Dom3_Dom2_Dom1	84.022
Dom4 Dom3_Dom2_Dom1	84.878
Dom4 Dom3 Dom2_Dom1	85.164
Dom4_Dom3 Dom2_Dom1	84.593

Table 2: Partial specialization. The underscore shows which domains were specialized, and which were left in generic form. For example, Dom2_Dom1 specializes Dom2 to make only calls from Dom1.

6.2. Maple

If we use Maple versions of our modules `dom1` and `dom2`, we can make a simple test of the potential for speed up for parametric polymorphism in Maple:

```
> dom1 := proc()
       module()
              local v;
              export a;
              a := proc(m::integer)
                     local i;
                     v := 0;
                     for i from 1 to 1000 do v := v + 1 end do;
                     return v;
              end proc:
       end module
end proc:

> dom2 := proc(p::`module`)
       module()
              local v;
              export s;
              s := proc(m::integer)
                     local i;
                     v := 0;
                     for i from 1 to 10000 do v := p:-a(1) end do;
                     return v;
              end proc:
       end module
end proc:
```

This can be specialized to:

```
> dom2_dom2_dom2_dom1_o := proc()
    module()
        local v;
        export s;
        s := proc(m::integer)
            local i,i1,i2,i3,v1,v2,v3;
            v := m;
            for i from 1 to 100 do
                v1 := v;
                for i1 from 1 to 100 do
                    v2 := v1;
                    for i2 from 1 to 100 do
                        v3 := v2;
                        for i3 from 1 to 10 do v3 := v3 + 1 end do;
                        v2 := v3;
                    end do;
                    v1 := v2;
                end do;
                v := v1;
            end do;
            return v;
        end proc;
    end module
end proc:
```

And the corresponding results are:

```
> d1 := dom2(dom2(dom2(dom1()))):
> t:=time(): d1:-s(1): time()-t;
                              10.312
> d3_o := dom2_dom2_dom2_dom1_o():
> t:=time(): d3_o:-s(1): time()- t;
                              6.438
```

As can be seen from the timings, this gives a speedup of about 36%.

7. Conclusion

We have observed that computer algebra programs that make use of parametric polymorphism tend to make heavy use of deeply nested constructions. Our experiments have shown that this leads to opportunities for significant code optimization. We have shown here straightforward test cases giving a speed up of 85%, for Aldor, and 36%, for Maple.

Our current tests using Aldor show significant improvements when specialization is applied to entire type constructors. Full specialization produces the fastest code in most of the cases, but it also produces considerable increase in size. We have observed that uniform specialization is not necessary, and we believe that heuristics that would produce fast code without full specialization. We would like to create more complex examples to see how they interact with the other optimizations provided by the Aldor compiler.

In Maple we have seen significant improvement for some simple examples. This shows that a specializer could be implemented for Maple, but it would also require the implementation of other optimizations to be most effective. There is considerable scope for improvement here. Most immediately, Maple's function inlining should be generalized to handle programs with local variables and multiple statements.

Code specialization is a well-known technique for performance improvement. Having the libraries implemented in the most generic form and then use a specializer like the one proposed here would offer the many advantages of smaller libraries.

8. Reference

[1]: M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, P. DeMarco, *Maple 9 Advanced Programming Guide*, Maplesoft, 2003

[2]: C. Oancea, S. M. Watt, *Domains vs Expressions: An efficient high-level interface between Aldor and Maple*, 2005,

[3]: Stephen M. Watt and Peter A. Broadbery and Samuel S. Dooley and Pietro Iglio and Scott C. Morrison and Jonathan M. Steinbach and Robert S. Sutor, *Aldor User Guide*, 2000, http://www.aldor.org/

[4]: Neil Jones and Carsten Gomard and Peter Sestoft, *Partial Evaluation And Automatic Program Generation*, 1995

[5]: Jeffrey Dean and Craig Chambers and David Grove, *Selective Specialization for Object-Oriented Languages*, 1995

[6]: Stephen M. Watt and Peter A. Broadbery and Pietro Iglio and Scott C. Morrison and Jonathan M. Steinbach, *FOAM: First Oder Abstract Machine*